Guida Python
>= 2.5

a cura di

Markon

Un grazie di cuore alla
comunità di Python-it.org.

Indice

1 Prefazione....2
1.1 Python FAQ....2
1.2 Perchè Python?....2
1.3 Come cominciare?....2
1.4 IDE o ambiente di sviluppo per Python?....3
1.5 Prerequisiti per imparare Python....3
2 Introduzione a Python....4
2.1 Cos'è un linguaggio di programmazione?....4
2.2 Cos'è Python?....4
2.3 Cosa vuol dire "interpretato"?....5
2.4 Una breve introduzione....6
3 Variabili e controlli....8
3.1 Iniziare con Python....8
3.2 Cos'è una variabile?....9
3.3 Definire le variabili....9
3.4 Ogni variabile ha il suo indirizzo....10
3.5 Tipi di dati....10
3.6 Effettuare controlli....11
4 Cicli e Iterazione....13
4.1 While....13
4.2 For....14
5 Crediti, Ringraziamenti, Licenza....17
5.1 Crediti....17
5.2 Ringraziamenti....17
5.3 Licenza....17

1 Prefazione

In queste pagine tratterò del linguaggio di programmazione Python.

Ogni capitolo tratterà un argomento e in più saranno mostrati alcuni esempi di codice con commenti.

Inoltre saranno disponibili degli esercizi per ogni sezione, così che possiate mettere alla prova ciò che avete appreso.

1.1 Python FAQ

Prima di cominciare la nostra trattazione, vorrei rispondere ad alcune delle domande più frequenti. Tuttavia le vere – e più complete – FAQ sono sul sito ufficiale di Python.

1.2 Perchè Python?

Innanzitutto, Python è un ottimo linguaggio di programmazione che presenta molte caratteristiche già implementate (builtin) come liste, dizionari e tanto altro ancora. Con Python è possibile interfacciarsi al sistema operativo tramite apposite librerie create dagli stessi sviluppatori del linguaggio.

E' un linguaggio che permette che il proprio codice possa essere letto da più macchine e da più sistemi operativi, grazie al fatto che è interpretato. Per tale motivo non dovrete preoccuparvi molto del fatto che il vostro programma possa girare su un sistema operativo o meno: basta che sia disponibile l'interprete Python per quel sistema operativo e sicuramente il vostro codice girerà.

Python sta crescendo soprattutto grazie al suo largo utilizzo in progetti Web e in particolare grazie a web framework come Django, Zope e simili. Tuttavia il suo ambito non è ristretto, tant'è vero che molti strumenti e utilty sono scritti in Python, grazie alla facilità e rapidità di sviluppo.

E' utile sviluppare in Python?

Certamente!

Python è uno dei linguaggi di programmazione attualmente più usati. E' il linguaggio di programmazione del 2007. E sta piano piano crescendo con la sua comunità. Infatti va notato che secondo LinuxQuestions.org Python è il linguaggio anche del 2008!

1.3 Come cominciare?

Va bene **qualsiasi** sistema operativo (anche se i file che userò in queste lezioni saranno tutti eseguiti su una macchina Linux) su cui sia possibile installare l'interprete Python, che in poche parole è ciò che vi permetterà di far girare i vostri programmi(ni).

Potete scaricarlo dal sito ufficiale, o al massimo dal sito ASPN.

Dunque lo scaricate e lo installate.

Per gli utenti Windows:

*per registrare le variabili d'ambiente (che vi permetteranno di eseguire python scrivendo **python** nel prompt dei comandi), potete utilizzare questo script. Qui è indicato come utilizzare lo script. Se invece preferite farlo senza l'utilizzo dello script, andate qui.*

1.4 IDE o ambiente di sviluppo per Python?

Per scrivere il codice (in poche parole i vostri programmi) avrete bisogno di un programma di scrittura. Va bene un qualsiasi editor di testo (es.: su Linux potete usare Kate o Gedit, su Windows potete usare il Blocco note o anche il Wordpad). Tuttavia esistono dei programmi che vi permetteranno, quando ne avrete bisogno, di gestire interi progetti, i vari moduli Python (poi vedremo cosa sono), di eseguire il programma che si sta scrivendo, di effettuare il debug (vedremo in seguito di cosa si tratta: per ora pensate che è un sistema per trovare gli errori). Questi programmi sono chiamati IDE (Integrated Development Environment).

Esistono **alcuni** ambienti di sviluppo per Python. Qui trovate una lista quasi completa degli editor per Python, mentre qui una lista degli IDE. La differenza principale tra i due è che con l'editor è possibile solo (nella maggior parte dei casi) scrivere il codice (il programma) e salvarlo, mentre un IDE ha delle caratteristiche che un normale editor di testo non ha. Personalmente vi consiglio questi editor:

- Linux : Kate o Gedit;
- Windows : Blocco Note o Notepad++ (free);
- MAC : Editor di testo predefinito ;

Potete notare in queste due immagini la differenza tra un semplice editor di testo e un IDE:

- Kate - (Editor di testo)
- Eric Python - (IDE)

1.5 Prerequisiti per imparare Python

Non c'è bisogno di alcun requisito per imparare Python. Tuttavia, come per ogni studio, avere già conoscenze pre-acquisite di altri linguaggi può portare i suoi vantaggi.

Vi auguro una buona lettura. Per ogni genere di problema, siete pregati di contattare l'autore della guida sul sito Markon's Blog.

2 Introduzione a Python

Dopo aver dato un'occhiata alle domande più frequenti, vediamo ora più da vicino cos'è Python e come è possibile utilizzarlo.

2.1 Cos'è un linguaggio di programmazione?

Una delle domande che naturalmente vi siete posti è stata questa. E non è una domanda banale. Per rispondere a tale domanda bisogna far capire che un computer funziona in un modo un po' diverso dal nostro. Noi umani in un modo o nell'altro siamo tutti uguali per cui altro non dobbiamo fare che studiare una nuova lingua o un nuovo linguaggio per poterlo utilizzare.

I computer vengono progettati con determinate strutture e implementazioni, così un processore Intel x86 sarà diverso da un MIPS, perchè sostanzialmente sono nati per motivi diversi e soprattutto perchè sono stati creati da aziende diverse.

Il primo linguaggio quindi che ogni computer ha è quello della sua macchina, ovvero l'assembler, che gli viene "insegnato" dagli ingegneri che hanno creato il processore. Tuttavia, poiché programmare in assembler è alquanto complesso, sono nati linguaggi più semplici, grazie all'evoluzione dei computer. Per questo motivo sono stati creati gli interpreti e i compilatori, che più avanti vedremo cosa sono. Ma alla base c'è sempre l'assembler.

Tuttavia ciò che è importante capire è che una qualunque istruzione in un linguaggio di programmazione come Python (indirettamente) o C (direttamente) viene tradotta letteralmente in assembler.

Un linguaggio di programmazione è ciò che ci permette di dire al nostro processore cosa deve fare.

2.2 Cos'è Python?

La definizione ufficiale è questa: è un linguaggio di programmazione interpretato, che supporta il paradigma object-oriented e in più è estendibile con altri linguaggi (potete scrivere dei file (chiamati librerie) in C o C++ e farli funzionare con Python).

Il suo creatore è Guido Van Rossum, che lo chiamò così grazie alla serie tv "Monty Python", che mi pare in Italia non sia mai stata trasmessa.

E' un linguaggio molto potente perchè uno dei suoi pregi è quello di avere le "batterie incluse", che vi permetteranno di concentrarvi su ciò che realmente dovete fare. Voi non dovete fare altro che imparare e capire cosa volete fare.

Così vi suggerisco di dare un'occhiata a queste "batterie", che vi aiuteranno a fare ciò che pensate, senza perdere tempo. Naturalmente per poterle utilizzare c'è bisogno di conoscere prima il linguaggio. Per ora dunque vi consiglio di aggiungere il link nei segnalibri del vostro browser web (Firefox ad esempio).

2.3 Cosa vuol dire “interpretato”?

La definizione di Wikipedia per un linguaggio interpretato è questa:

> *In informatica, un **interprete** è un programma che esegue altri programmi.*
>
> *Un **linguaggio interpretato** è un linguaggio di programmazione i cui programmi vengono eseguiti da un interprete. Tale approccio si distingue da quello dei linguaggi compilati: a differenza di un interprete, un compilatore non esegue il programma che riceve in ingresso, ma lo traduce in linguaggio macchina (memorizzando su file il codice oggetto pronto per l’esecuzione diretta da parte del processore).*

Dunque c’è una differenza tra linguaggi interpretati e compilati.

Python è un linguaggio interpretato, come il Perl, Bash, (altri linguaggi interpretati) etc… perchè necessita di un interprete, un programma che legge il codice sorgente e lo esegue, mentre i linguaggi compilati compilano ciò che voi scrivete in linguaggio macchina.

Immaginate ora questa situazione: scrivo un programma che deve essere compilato (in linguaggio macchina) e un mio amico mi chiede di passargli il programma. Questo mio amico dimentica però di dirmi un particolare a dir poco fondamentale: ha un PowerPC, mentre io ho solo un x86. Cosa succederà secondo voi? Semplice, la sua macchina non saprà eseguire quelle istruzioni.

La stessa cosa non si può dire per un linguaggio interpretato, che grazie al programma che esegue il codice che abbiamo scritto, non ha problemi a girare su computer diversi.

Perchè avviene questo? Ebbene, l’interprete, che esegue **ogni** riga del nostro codice, varia a seconda del computer (mi riferisco sempre al processore), per cui ciò che noi scriviamo nel nostro programma viene interpretato in base all'architettura.

In questo modo garantisce ciò che comunemente viene chiamata **portabilità**, ovvero la possibilità, come abbiamo visto, di poter far funzionare su tutti i computer che abbiano installato l’interprete il nostro programma.

Dunque un linguaggio interpretato può essere un vantaggio come può essere uno svantaggio:

- è codice portabile(può essere eseguito su Windows, su Linux, su Mac, etc.., senza cambiare nulla nel codice), perchè c’è *un solo* interprete;
- può risultare più lento del codice compilato, perchè viene interpretata ogni singola istruzione;

Immagine che rappresenta il codice interpretato:

Immagine che rappresenta il codice compilato:

Per ora queste sono le cose più importanti da sapere.

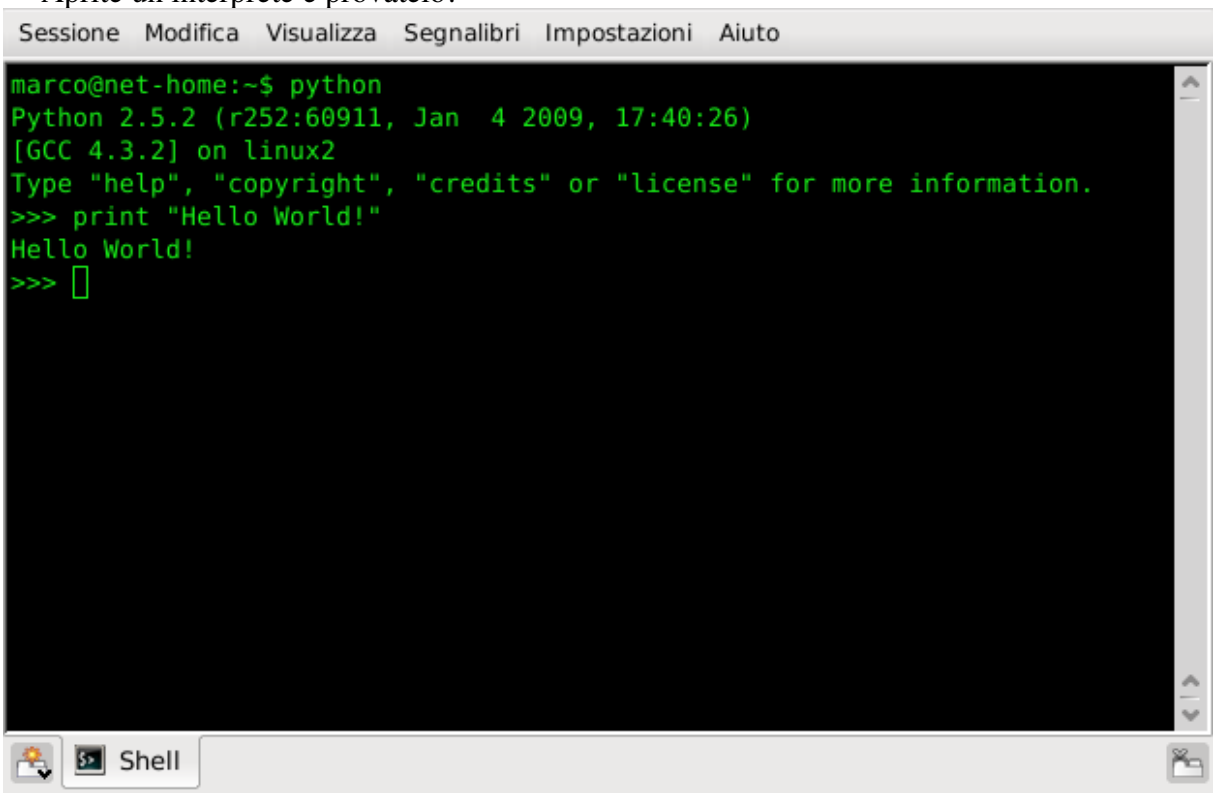
2.4 Una breve introduzione

L'esempio classico che troverete sul web o in un libro sulla programmazione è quello dell' "Hello world".

Così cominceremo con questo esempio:

```
print "Hello World"
```

Aprite un interprete e provatelo!



E' davvero emozionante, lo so, soprattutto se è il vostro primo linguaggio di programmazione.

Così avete visto come con la parola *print* potete "stampare come output" cosa volete.
Potete stampare anche il vostro nome o ciò che desiderate di più, cambiando la parola "Hello World" con ciò che volete. Come potete notare non c'è bisogno di altre aggiunte: tutto ciò

che vi serve per stampare una semplice parola (più precisamente si chiama “stringa”) è scritto lì.

Questo capitolo termina qui. Era solo una breve introduzione a Python!

3 Variabili e controlli

In questo capitolo cominciamo a vedere come funziona Python. Affronteremo il concetto di *keyword* (parola chiave), *variabile*, di come si definisce una variabile e a cosa serve sostanzialmente.

3.1 Iniziare con Python

Nel capitolo precedente abbiamo visto il classico esempio "Hello world", che utilizzava una *keyword* (parola chiave) di Python (si tratta della parola *print*).
Vi chiedete a questo punto cosa sia una keyword?
Sappiate che le keyword sono delle parole presenti in ogni linguaggio di programmazione (non sono tutte uguali, ma molte sono identiche) e che hanno uno scopo ben definito.

Possono essere paragonate alle congiunzioni della nostra lingua, dato che ognuna di esse crea una frase subordinata. Solo che qui non utilizziamo le virgole, i punti e virgola, i punti, ma altri tipi di punteggiatura.
Il loro scopo è quindi quello di creare una frase subordinata, che molto spesso è utile perchè come nella normale lingua è importante non parlare per frasi semplici come "*Sono andato al cinema. Ho guardato un bel film. Sono tornato a casa. Ho mangiato e ho dormito*" (frase che poteva essere scritta come "*Dopo essere andato al cinema, in cui ho guardato un bel film, sono ritornato a casa, dove ho mangiato, dopo di che sono andato a dormire*"), così nella programmazione è importante saper creare frasi più complesse.

Una domanda: nella nostra lingua è possibile utilizzare queste congiunzioni così a caso? E' possibile mentre scriviamo una lettera o un tema mettere queste parole a caso? Pensate alla frase:

> *Ciao, come poichè si chiama che o che se si dice nonostante?*

E' grammaticalmente corretta, ma sintatticamente scorretta: la frase corretta sarebbe stata "Ciao, come si chiama? Che si dice?".
Ebbene, come nella nostra lingua, così anche nel nostro linguaggio di programmazione bisogna utilizzare una sintassi corretta.
Dobbiamo dunque immaginare che quando scriviamo un programma ci sia un professore bravissimo che non sbaglia mai e che corregge ciò che abbiamo scritto. Nel caso in cui ci dovesse essere un errore il professore ci segnala l'errore. Guardate questo esempio:

```
>>> if = 20
  File "<stdin>", line 1
    if = 20
       ^
SyntaxError: invalid syntax
```

Cosa stiamo cercando di fare? Stiamo cercando di dare il valore 20 a una parola chiave. Come nella nostra lingua non è possibile cambiare il significato delle congiunzioni, così nella programmazione non è possibile cambiare il significato (si chiama *valore*) alle parole chiave. Così il nostro professore (che altri non è che l'*interprete*) ci segnala subito l'errore. (meglio

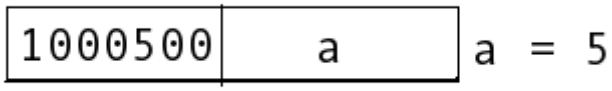
no?) La cosa ancora più interessante è che ci dice chiaramente che errore abbiamo fatto: in questo caso un *SyntaxError*!!

3.2 Cos'è una variabile?

Innanzitutto "definiamo" il significato di variabile:

> *In informatica, una variabile identifica una porzione di memoria destinata a contenere dei dati, che possono essere modificati nel corso dell'esecuzione di un programma.(Wikpedia)*

Dunque una variabile altro non è che ciò che viene registrato nella memoria del nostro computer. Come avviene questo?
Pensiamo ai diversi tipi di oggetti esistenti in un linguaggio: esistono le lettere, i numeri interi, i numeri con la virgola, e così via. Ora immaginate la memoria del nostro computer come una tabella.



Nell'immagine possiamo vedere che la variabile definita con il valore 5 si trova nell'indirizzo 1000500 della memoria (ovviamente su un computer diverso si trova in un indirizzo diverso).
Python semplifica le cose gestendo da solo gli indirizzi della memoria e non permettendo all'utente di assegnare determinati valori ad un indirizzo della memoria (cosa che spesso risulta di difficile comprensione e che porta ad errori).

3.3 Definire le variabili

Ora vediamo meglio come si definisce una variabile in Python.

> *E' preferibile usare il termine "definire" a "dichiarare", perchè mentre il primo assegna direttamente il valore alla variabile, il secondo si limita a dichiararne il tipo e il nome. Come vedremo, in Python basta assegnare un valore a un nome (variabile)*

Aprite un terminale e divertitevi a definire variabili:

```
>>> a = 10
>>> type(a)
<type 'int'>
>>> b = "Io sono una stringa"
>>> type(b)
<type 'str'>
```

Ora provate voi a cambiare i valori alle stesse variabili già definite o a definire nuove variabili.
Vi chiederete da dov'è uscito *type*. Ebbene, *type* è una di quelle funzioni che rappresentano

parte delle "*batterie*" di cui abbiamo parlato nella lezione precedente. Essa non fa altro che *ritornare* il tipo della variabile (come vedete varia da "int" a "str"…, dove "int" sta per intero, "str" sta per stringa, e così via…).

Se avete provato a definire delle variabili probabilmente avrete ricevuto qualche errore, come *SyntaxError*. Questo perchè Python ha delle regole (come abbiamo visto) anche per i nomi delle variabili: il nome deve cominciare con una lettera (o con un underscore, il segno "_") e può contenere solo lettere e numeri (ad esempio var1 = 10).
Per cui se avete provato a utilizzare caratteri come *&*, *%*, *£*, e simili, avrete ricevuto un errore di *sintassi*.

3.4 Ogni variabile ha il suo indirizzo

Ricorderete che nella pagina precedente abbiamo visto che Python è un linguaggio interpretato. Uno dei suoi vantaggi è proprio nella definizione delle variabili.
Innanzitutto Python usa ciò che viene comunemente chiamata "tipizzazione dinamica": è possibile definire una variabile con un valore di un certo tipo (ad esempio intero, *int*) e cambiarlo ogni qual volta vogliamo in qualsiasi tipo desideriamo (ad esempio in stringa).

Siamo poi sicuri che il valore corrispondente all'indirizzo di memoria cambi una volta riassegnato un altro valore alla stessa variabile?
Verifichiamolo con la funzione *id*!

```
>>> a = 10
>>> id(a)
135712460
>>> a = 20
>>> id(a)
135712340
```

Come potete notare, i due valori, meglio definiti come "indirizzi" nella memoria, sono diversi e ciò significa che la variabile è stata distrutta e ricreata: naturalmente i valori possono cambiare da computer a computer, a seconda della memoria concessa dal sistema operativo.

3.5 Tipi di dati

Ogni variabile può contenere dei valori. Ma questi valori non devono essere necessariamente "stringhe" o "interi" o "numeri decimali" (chiamati *float*). Python mette a disposizione altri tipi di dati:
liste, dizionari, tuple, insiemi e booleani.
Ma vediamo meglio:

- I numeri: come nel linguaggio comune, anche in Python esistono i numeri.
- Le *stringhe* sono per definizione una sequenza di caratteri, *immutabili*;
- Una *lista* è un insieme *mutabile* che può contenere elementi di diverso tipo (possiamo mettere insieme stringhe, interi, istanze di classe, e così via);

- Gli insiemi sono un tipo di dato non ordinato contenente oggetti *non* duplicati al suo interno;
- Le *tuple* sono sequenze di oggetti che possono essere eterogenei (come per le liste), ma sono *immutabili*;
- I *dizionari* sono un insieme di oggetti *chiave/valore*, i cui valori sono ottenuti attraverso le *chiavi*;
- I *booleani* sono un tipo di dato comune nei linguaggi di programmazione: essi sono *True* e *False*;

Questi tipi di dati li useremo in tutto il tutorial. Se vi interessa vedere qualche esempio di come si usano, basta che facciate clic sui link a disposizione.

E' interessante ricordare che la gestione della memoria viene lasciata a Python, così il programmatore non deve più preoccuparsi di creare un nuovo tipo di dato per numeri enormi, perchè a questo ci pensa già l'interprete.

Infatti, come molti programmatori C sanno, un intero generalmente è contenuto in 4 byte (32 bit) e riesce a rappresentare massimo 4'294'967'295. Il programmatore Python dunque non deve preoccuparsi dei limiti dei suoi numeri.

3.6 Effettuare controlli

Dopo aver dato uno sguardo alle variabili in Python, che come abbiamo visto sono dinamiche, ora possiamo proseguire dando uno sguardo ad alcune delle parole chiave.

Come abbiamo visto le parole chiave sono come le congiunzioni, per cui è possibile utilizzarle per determinati scopi.
Inizieremo a vedere la keyword *if*, che tradotta dall'inglese vuol dire "se".
Uno dei punti forti di Python è proprio questo: il fatto che sembri quasi di stare lì a scrivere in inglese, mentre invece si sta creando il nostro progetto di lavoro.
Vediamo ora come funziona *if*.

```
>>> a = 10
if a == 10:
... print("a = %d" %10)
...
a = 10
```

Se abbiamo bisogno di effettuare maggiori controlli, ci sono *elif* (è l'unione delle parole "else if": altrimenti se) e *else*(altrimenti).

```
>>> if a < 20:
...     print("%d e` minore di 20" %a)
... elif a > 20:
...     print("%d e` maggiore di 20" %a)
... else:
...     print("%d e` uguale a 20" %a)
10 e` minore di 20
```

Vi starete chiedendo cos'è il simbolo '%'. Vedremo in seguito di cosa si tratta, per ora sappiate che serve a formattare le stringhe.

Come vedete, alla fine di ogni controllo vanno messi i due punti (:); come vedremo sono necessari anche per gli altri cicli, per la definizione di funzioni, classi, etc…
Una cosa importante che avrete notato (se conoscete già altri linguaggi di programmazione) è che Python a differenza di molti altri linguaggi di programmazione non termina le proprie istruzioni (ciò che deve fare il programma, ovvero ogni riga di codice [a parte i cicli di controllo]) con il punto e virgola (;).

E' possibile utilizzarlo ma è inutile

```
a = 10 ; # l'interprete non riporterà alcun errore
```

Ora è dunque abbastanza chiaro perchè Python abbia sollevato il famoso *SyntaxError* a cui abbiamo accennato all'inizio della pagina. Ma per chi non l'avesse ancora intuito:
dopo l'if (o altre istruzioni, come "elif", ..) va messa un'espressione di controllo.

4 Cicli e Iterazione

Dopo aver visto cosa sono le variabili e alcune delle parole chiave per capire come effettuare un controllo, in questo capitolo vedremo come produrre un ciclo.

Secondo il teorema di Bohm-Jacopini:

> *Il **teorema di Böhm-Jacopini**, enunciato nel 1966 dagli informatici Corrado Böhm e Giuseppe Jacopini, afferma che qualunque algoritmo può essere implementato utilizzando tre sole strutture, la **sequenza**, la **selezione** ed il **ciclo**, da applicare ricorsivamente alla composizione di istruzioni elementari (ad es. di istruzioni eseguibili con il modello di base della macchina di Turing)*

Finora abbiamo visto, seppure molto genericamente, la prima e la seconda parte. Ora ci tocca affrontare la terza parte - il **ciclo**.

Mentre della **sequenza** fanno parte le istruzioni singole in "sequenza" - appunto - e della **selezione** fanno parte le istruzioni che determinano la scelta, nonchè i controlli su variabili e così via, del **ciclo** fanno parte le istruzioni che permettono di ripetere il codice, perchè il lavoro del programmatore è in primis quello di non ripetere due volte una medesima cosa.

Come è possibile dunque effettuare un **ciclo** - o *iterazione* - in Python?

Ebbene, in maniera molto simile ad altri linguaggi di programmazione, è possibile effettuare dei **cicli** con delle parole chiave – noi le abbiamo paragonate alle congiunzioni – che ci permetteranno di ripetere delle istruzioni per tutto il tempo che si verifica una determinata **condizione**.

4.1 While

while - da scrivere sempre in minuscolo, come tutte le altre parole chiave - è una keyword (parola chiave) che può essere tradotta come "*finchè*". Che frase potrebbe produrre una congiunzione simile?

Un esempio potrebbe essere:

"Finchè non smetterà di piovere, noi rimarremo a casa".

Proviamo a immaginare una cosa del genere in un nostro programma. Per tutto il tempo che si verifica una condizione da noi stabilita il programma deve svolgere delle istruzioni.

La sintassi, ovvero il modo con cui dobbiamo generare il **ciclo** è simile a quella utilizzata in precedenza per i controlli:

```
while (condizione):
    codice da ripetere ...
```

> *N.B.: Non è obbligatorio inserire la condizione tra parentesi, tuttavia è consigliato soprattutto per garantire maggiore leggibilità e evitare errori di logica.*

In teoria quindi si potrebbe rappresentare “Pythonicamente” la frase sopra descritta in questo modo:

```
while (pioggia > 0):
    rimani_a_casa = True
    pioggia = pioggia - 1 # dovrà pur smettere di piovere no?! :)
```

Non ha molto valore un codice del genere, però può essere rappresentato proprio così.

Ora vediamo da vicino come funziona un **ciclo while**.

```
x = 0
while (x < 5):
    x += 1
```

Come potete notare e come si può intuire, non facciamo altro che inizializzare una variabile a 0. A cosa ci serve quindi il **ciclo while** ?

In questo caso serve per verificare che la *x* sia minore di 5. Funziona un po’ da “contatore”. Infatti, finchè la **x** è minore di 5, allora aggiungiamo 1 ad essa, finchè non si verifica la condizione per la quale x arriva a 5. In tal modo la **condizione** tra parentesi risulta falsa e il ciclo si ferma.

Il ciclo while quindi non fa altro che verificare innanzitutto che la condizione sia vera. Se lo è, allora procede con le istruzioni di seguito, altrimenti si ferma o non prosegue proprio.

Potreste chiedervi ora: “Che cosa succede se la condizione non risulta mai falsa?”

Ebbene, il ciclo continua all’infinito.

Spesso questo può essere causato da un errore di logica, che può portare ad errori nel programma o ad una chiusura inattesa del programma. Quando accade questo in genere si dice che “va in loop”.

Tuttavia, ci sono dei casi in cui è utile generare un ciclo infinito, che ovviamente deve essere controllato dal programmatore, così che non sia generato alcun tipo di errore inatteso.

4.2 For

In Python esiste anche un altro ciclo per “iterare”, che prevede l’uso di un’altra parola chiave: il ciclo *for*.

Come funziona questo ciclo *for*?

La sintassi del *for* è la seguente:

```
for elem1, ... in obj1, ...:
    ...istruzioni...
```

Ebbene, questo ciclo, a differenza del precedente, *itera* su una o più sequenze di oggetti, ad esempio numeri.

Immaginiamo infatti di voler stampare tutti i numeri da 0 a 100. Come potremmo fare?

O usiamo un ciclo while con una variabile *contatore* (inizializzata a 0 magari), che ad ogni ciclo aumenta "+1″ alla variabile *contatore* e la stampa, o utilizziamo una funzione che ci viene data dal linguaggio stesso. Ricordate quelle "*batterie incluse*"? Ecco, la funzione "range" è un'altra di queste.

Volete sapere cosa fa la funzione range? Basta che avviate l'interprete Python e che lanciate il comando "*help(range)*" per ottenere questo:

```
range(…)
range([start,] stop[, step]) -> list of integers
Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

Per uscire dalla modalità di help, basta che digitiate la lettera '*q*'.

La funzione prende almeno un argomento (quelli opzionali si trovano fra parentesi quadre) per un massimo di 3 argomenti. Come è intuibile, l'argomento obbligatorio è quello al centro, "stop". Ritorna una lista di interi, come potete notare dalla freccetta (→)

Cos'è una funzione e cosa sono gli argomenti e parametri lo scoprirete presto. Per ora sappiate che quando richiamate una funzione che prende un argomento, dovete inserire un elemento tra le parentesi. Questo elemento naturalmente non può essere di "qualsiasi" tipo, ma di un tipo ben definito. Per ora però non mi dilungo troppo nella spiegazione dei tipi e degli argomenti. Per ora è importante tenere in mente che una funzione che chiede almeno un argomento, ha bisogno di ricevere un elemento tra parentesi.

Come possiamo quindi stampare i primi 100 numeri, partendo da 0, con un ciclo for?

La risposta è la seguente:

```
for x in range(100):
    print(x)
```

L'output sarà il seguente:

```
0
1
2
.. # numeri omessi
98
99
```

Dovrebbe essere abbastanza chiaro a questo punto a cosa servano i cicli in un linguaggio di programmazione. Ma se non lo è ancora, allora sappiate che servono per far sì che possiate scrivere un programma che faccia qualcosa per voi senza che utilizziate tante righe di codice. Infatti se avessimo voluto stampare tutti i numeri da 0 a 100 senza usare il ciclo for (o anche

senza il ciclo while) avremmo dovuto scrivere 100 volte "*print(..)*" dove al posto dei puntini c'era il numero che volevamo stampare.

Spero dunque che sia chiaro finora a cosa servono le variabili, le condizioni e i cicli.

Esercizi:

- Stampare tutti i numeri pari presenti da 0 a 100 (quindi 0, 2, 4, …, 98)
- Stampare tutti i numeri pari, moltiplicati per 2, presenti da 0 a 100. (questo esericizio altro non è che lo stesso di sopra, con una piccola aggiunta).

5 Crediti, Ringraziamenti, Licenza

5.1 Crediti

Autore del tutorial: Marco Buccini <aka Markon : buccini.marco@pythonmark.com>

5.2 Ringraziamenti

Vorrei ringraziare lo staff di Python-it.org e tutti gli sviluppatori di Python.org

5.3 Licenza